Validation process for railway interlocking systems

Validation Process for Railway Interlocking Systems

A. Bonacchi[a], A. Fantechi[a], S. Bacherini[b], M. Tempestini[b]

[a] *DINFO - University of Florence, Via S. Marta 3, Firenze, Italy*
[b] *General Electric Transportation Systems, Firenze, Italy*

Abstract

An interlocking system monitors the status of the objects in a railway yard, allowing or denying the movement of trains, in accordance with safety rules. The high number of complex interlocking rules that guarantee the safe movements of independent trains in a large station makes the verification of such systems a complex task, which needs to be addressed in conformance with EN50128 safety guidelines.

In this paper we show how the problem has been addressed by a manufacturer at the final validation stage of production interlocking systems, by means of a model extraction procedure that creates a model of the internal behaviour, to be exercised with the planned test suites, in order to reduce the high costs of direct validation of the target system.

The same extracted model is then subject to formal verification experiments, employing an iterative verification process implementing slicing and CEGAR-like techniques, defined to address the typical complexity of this application domain.

Keywords: Railway Interlocking Systems, System Validation, Model-based Testing, Formal Methods, Model Checking

1. Introduction

Among railway signalling systems, the *interlocking* systems are responsible of allowing or denying, according to established safety and operational regulations, the routing of the trains through stations or railway yards. Safety

Email addresses: a.bonacchi@unifi.it (A. Bonacchi), alessandro.fantechi@unifi.it (A. Fantechi), stefano.bacherini@ge.com (S. Bacherini), matteo.tempestini@ge.com (M. Tempestini)

and operational regulations are generic for the region or country where the interlocking is located, and their instantiation on a particular station or yard topology is usually captured by a so called *control table*. Control tables of modern computer-based interlocking systems are implemented by means of iteratively executed software tests over the status of the yard objects.

One of the most common ways to describe the interlocking rules given by control tables is through boolean equations or, equivalently, ladder diagrams which are interpreted either by a Programmable Logic Controller (PLC) or by a proper evaluation engine over a standard processor.

In particular, in (7), we have introduced an effort made in a cooperation between the University of Florence and the Safety and Validation (S&V) team of General Electric Transportation Systems (GETS) in Florence, with the final aim of reducing the costs of verifying the safety requirements of the produced interlocking systems, in the system validation phase.

The S&V team performs the final independent validation, according to CENELEC EN50128 standard (9), of the produced railway signalling systems, and hence it acts as an independent verifier of the interlocking systems produced by other branches of the company, with little insight of the followed development process, and focusing on the final product.

According to the separation between design teams and validation teams, the information accompanying a system that undergoes validation is constituted solely by the controlled station layout, and by a set of test suites defined by the signalling engineers for that particular system.

To gain more insight over the control tables encoded within the system under evaluation, it is possible to extract those control tables by means of proprietary procedures (called in the following *legacy libraries*) that provide the control tables in a proprietary format. For confidentiality reasons, the format cannot be disclosed, nor any fragment of extracted control tables.

Hence, the scope of our work is the modelling of the control tables extracted from the binary files, by means of a *reverse engineering* process, in order for the tests suites to be simulated on the model; the expected advantages are expected to be given by the early validation of the implemented control tables, before a full physical test bench for the specific equipment is built.

The choice of the modelling tool was taken according to specific constraints posed by the S&V team: in order to smoothly adopt this verification technique inside the internal production process, a commercial development/verification tool, already known within the company, was a require-

ment. This constraint has favoured the choice of Matlab and *Simulink*, using Simulink logic gates to encode boolean functions extracted from the legacy control tables.

Moreover, in (6) we set up a verification framework based on model checking on the extracted model, employing Matlab *Design Verifier* (33), both because it works on Simulink models and to exploit at best its SAT-solving capabilities on the native boolean coding of the control tables. The verification framework exploits environment abstraction, slicing and CEGAR-like techniques, driven by the detailed knowledge of the interlocking product under verification.

The paper extends results from (5; 6; 7), which separately described the preliminary application of the proposed model-based testing and formal verification concepts. The two concepts are now jointly presented in an improved way, with more up-to-date details, reporting also how the former is already consolidated into the industrial validation procedure, while the latter is still at an experimental stage.

The paper is organized as follows: Section 2 presents the current activity of system testing in the S&V Laboratory of GETS, placing it in the context of the overall development cycle of safety critical systems in accordance with EN50128. In Section 3 we introduce Ladder Logic, which is used to implement control tables, and we introduce the model extraction process that allows the control tables to be translated into boolean functions implemented in a Simulink model. Section 4 introduces the testing made on the extracted models, showing the results obtained when three example interlocking systems are simulated with given simulation scenarios. In Section 5 we address formal verification discussing in particular the proposed slicing and abstraction techniques. Section 6 concludes the paper.

2. The role of validation in EN50128

CENELEC EN50128 is the standard that specifies the procedures and the technical requirements for the development of programmable electronic devices to be used in railway control and signalling protection (9). This standard is part of a family, and it refers only to the software components and to their interaction with the whole system. The basic concept of the standard is the *SIL* (Safety Integrity Level). Integrity levels characterize software modules and functions according to their criticality, and range is defined from 0 to 4, where 0 is the lowest level, which refers to software

functions for which a failure has no safety effects and 4 is the maximum level, for which a software failure can have severe effects on the safety of system, resulting in possible loss of human life.

2.1. Model-based design

The standard encourages the usage of models and formal methods in every phase of the development cycle for software of the highest SILs, starting from the design to the verification. The rationale is that models are more related to abstract concepts than the technologies used for their implementation into code, and are therefore closer to the domain of the problem. On the other hand, sufficiently detailed models can be used for automatic generation of code: automatic code generation, according to EN50128, requires that the code generator itself is somehow trusted.

In the past years GETS has committed to model-based design, by developing an internal software production process compliant with EN50128, employing the most advanced software production and verification techniques, such as formal methods, model-based design and testing, automatic code generation (15), static analysis based on abstract interpretation (16). Several insights into this process can be retrieved in (2); such a process includes a thorough verification activity on software units (16).

2.2. Independent validation

The EN50128 standard proposes three kinds of preferred organizational structures, according to the SIL, for the software development.

The eight roles (Project Manager, Requirement Manager, Designer, Implementer, Integrator, Tester, Verifier and Validator), defined by the standard for SIL3 and SIL4 software (to which we are interested in this context), should be fulfilled by at least five persons, in the same company, who shall (solid line) or can (dotted line) report their activity to the Project Manager (see Figure 1).

The Validator (VAL) shall not report to the Project Manager but informs him about his activity. Finally, the Assessor (ASR) is an independent entity that carries out the process of the analysis to determine whether the software meets the specified requirements and is fit for its intended purpose.

Note that the Verifier (VER) is part of the design and development team, and performs early verification tasks (including e.g. unit testing and structural testing). Validation is instead intended as conducted on the final products, by means of system testing, functional testing and black-box testing,

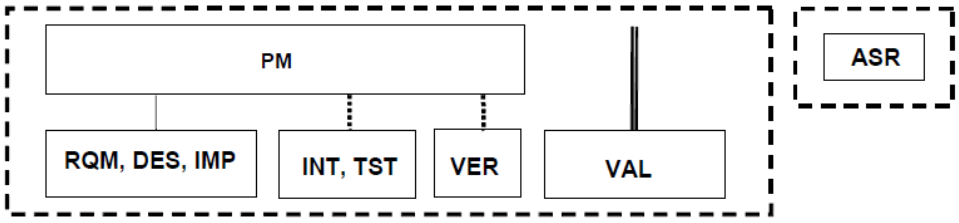

Figure 1: Roles in a company for SIL3 and SIL4 software

or so-called hardware-in-the-loop testing (8) of the actual installation. In accordance with EN50128, the S&V team of GETS acts as the Validator entity with the responsibility to carry out the validation testing of the railway signalling equipment produced by the different development teams of GETS.

2.3. Certification of specific applications

The EN50128 guidelines distinguish between generic applications and specific applications, defining generic software as "software which can be used for a variety of installations purely by the provision of application-specific data and/or algorithms". A Specific Application Software is defined as a Generic Application Software plus configuration data, or plus specific algorithms. These definitions clearly apply to interlocking systems: control tables act as configuration data (since they could be expressed as Ladder Logic programs, as seen in Sect. 3.1, they can be seen as specific algorithms as well).

EN50128 defines requirements about the validation of generic and specific applications. For example, *"Care must be taken in the verification and validation phases of the generic software in order to assure that all relevant combinations of data and algorithms are considered. If all relevant combinations of data and algorithms have not been considered, it shall be clearly identified as a limit of use of the generic software"*. For what concerns a specific application, evidence is requested that the application conditions of the generic application are met by the configuration data or the specific algorithms.

Despite various "shortcuts" being provided, the certification of a specific application, the application of which has already been certified as generic, requires only a little less effort than the certification of an ad hoc application. This means that the validation of any new installation of an interlocking system needs all the validation effort to be repeated on that installation, although it can focus on the correctness of control tables, assuming that the

generic characteristic of the interlocking system have been already validated for previous installations.

The hardware-in-the-loop testing requires lengthy scenarios to be run on the actual installation reproduced in the validation labs, and this must be repeated for different installations of the same system, since the control tables are different for each station. This makes validation a lengthy and costly process. In the following sections we investigate model-based validation methods that allow time and cost to be spared, by focusing on the specific source of differences between interlocking installations, that is, control tables.

According to CENELEC guidelines, this corresponds to concentrating on the validation of the configuration data of a specific applications rather than on the generic application (that is, the control table evaluation engine) which is taken as already validated.

3. Model Extraction

3.1. Ladder Logic Diagrams

In legacy relay-based interlocking systems, still operating in several sites, the logical rules of the control tables were implemented by means of physical relay connections. With modern computer-based interlocking systems, in application since 30 years, the control table becomes a set of software equations that are executed by the interlocking[1]. Since the signalling regulations of the various countries were already defined in graphical form for the relay-based interlocking systems, and also in order to facilitate the representation of control tables by signalling engineers, the design of computer-based interlocking systems has usually adopted traditional graphical representations such as *Ladder Logic Diagrams* (LLD) (27) and relay diagrams (21). These graphical schemata, also called *principle schemata*, are instantiated on a station topology to build the control table, which is then translated into a program for the interlocking.

Correctness of control tables depends also on the model of execution by the interlocking software. In building computer-based interlocking systems, the manufacturers adopt the principle of *as safe as the relay-based equipment*

[1]Indeed, we should rather talk about "assignment of boolean expressions to boolean variables" instead of "equations". However, the syntactic appearance as an equation system has favoured the common adoption of the latter term in the railway signaling community, and we therefore adopt this term consistently throughout the paper.

(35), and often the implemented model of execution is very close to the hardware behaviour of the latter.

Ladder logic is a graphical language which can represent a set of boolean equations and their execution order (*control cycle*). The control cycle can be detailed as the following equation system:

$$\tilde{x} = f(\tilde{x}, \tilde{y})$$

where $\tilde{x}, \tilde{y}$ are boolean variable vectors representing respectively state/output variables and input variables: these equations are sequentially and cyclically executed, where for "execution of an equation" we intend the assignment to its left hand of the evaluation of the right hand expression.

Ladder Logic represents the working of relay-based control systems. For this reason the variables on the right-hand side of the equation are also named *contacts*, while the variables in the left hand are named *coils*. Variables can be distinguished in:

- *Input variables*: the value is assigned by sensor readings or operator commands. These variables are defined in the expressions and cannot be used as coil.
- *Output variables*: can only be coils and their value is determined by means of the assignments of the diagram and is delivered to actuators.
- *Latch variables*: the value is calculated by means of the assignments, but is used only for internal computation of the values of other variables. A latch variable is used as coil in an assignment and as an input variable in other assignments.

With these three kinds of variables, a Ladder Logic Diagram (for short, a ladder diagram) describes a state machine whose memory is represented by the latch variables and whose evolution is described by the equations. An execution cycle of this state machine, named *control cycle*, involves:

1. Reading input variables; the values of these variables are assumed to be constant for the entire duration of the control cycle.
2. Computing each equation, in sequence, hence assigning values to the output variables and to the latch variables as a function of the current values of the input variables and the values of the latch variables computed by the previous control cycle.

3. Transmission of the values of the output variables.

In this way, the equations can be seen as interpreted by a reasoner engine. The reasoner engine is the same for every station plan, and hence it constitutes the *generic application* according to EN50128; the control table is coded as configuration data, actually boolean equations, for the reasoner. This approach is also referred as "data-driven". Behind this choice is the minimization of certification efforts: the reasoner is certified once for all, the data are considered "easier" to certify if they can be related in some way to the standard principle schemata adopted by railway engineers in the era of relay-based interlocking systems.

Ladder diagrams have a text-only syntax, where, if x is a boolean variable, an expression e can be defined as:

- "`--]  [--`" represents the reference to a variable.
- "`--]/[--`" represents the negation of a variable.
- "`--(  )`" represents a coil.
- To mimic a logical *and* two variables are wired in series.
- To mimic a logical *or* two variables are wired in parallel.

Fig. 2 shows an example of ladder diagram according to such syntax.

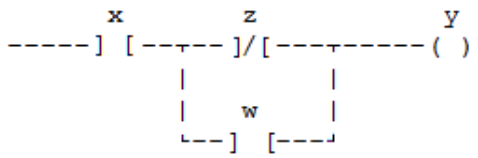

Figure 2: Equation $y = x \wedge (w \vee \sim z)$ as a ladder diagram

3.2. Model extraction process

The ladder diagram description of control tables is stored in the implemented system, so we had first to define a process that allows a model of a station to be obtained from the analysed implementation in two steps (see Figure 3):

1. **Import Station Data**: all data about a station (equations, timers, interfaces, ...) are imported in Matlab by means of proprietary *legacy libraries* that read the binary files loaded on the interlocking system.
2. **Model Station Data**: the equations are modelled in a Simulink model by means of a tool named *LLD-Parser* (7).

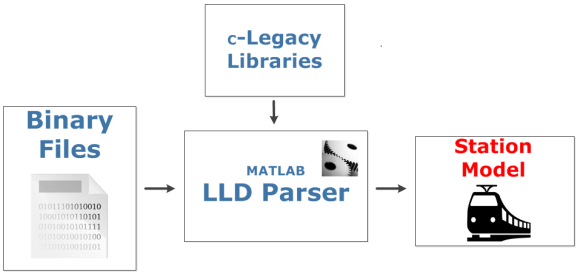

Figure 3: Model extraction process

3.3. Importing Station Data

In order to extract the information from the binary files, we use those proprietary interpretation routines that we have called *legacy libraries*. These libraries allow to read each boolean equation, written as a ladder diagram, as a matrix $M^{n\times k}$. The values in the matrix can be either positive integers, representing variables identifiers, or negative integers, representing either a connector or the polarity, asserted or negated, of a variable (see Table 1). The ladder diagram in Figure 2 is for example encoded by the following matrix M:

$$\begin{bmatrix} -40 & 100 & -40 & 200 & -40 & 500 \\ -50 & -10 & -4 & -20 & -4 & -30 \\ -40 & -40 & -70 & 300 & -70 & -40 \\ -40 & -40 & -1 & -10 & -2 & -40 \end{bmatrix}$$

The values 100, 200, 300 and 500 are respectively associated to the variables x, z, w and y.

3.4. LLD Parser

The extracted matrix, that represents a single equation, needs to be interpreted in order to define the boolean function it implements, expressed in a format suitable for Design Verifier.

Table 1: Symbol Translation

Symbol	Value	Symbol	Value
$\lfloor$	-1	$--[\quad]--$	-10
$\rfloor$	-2	$--[/\,]--$	-20
$\perp$	-3	$--(\quad)$	-30
$\top$	-4	Blank space	-40
$\vdash$	-5	Horizontal line	-50
$\dashv$	-6	Vertical Line	-70
$+$	-7		

We have hence designed an algorithm, sketched in Algorithm 1, that translates the matrix into a Simulink boolean function expressed as a combination of logic gates.

Algorithm 1 LLD Parser

Require: $M^{n\times k}$ equation matrix
Ensure: Model of the equation
1: $var \leftarrow$ GetVariables(M)
2: **if** $\top \in M$ **then**
3: $\quad LogicAnd(M, var)$
4: $\quad$ **for** i = 1 to k **do**
5: $\quad\quad LogicOr(M, var, i)$
6: $\quad\quad LogicAnd(M, var)$
7: $\quad$ **end for**
8: **else**
9: $\quad LogicAnd(M, var)$
10: **end if**

If we focus on the graphical format of ladder diagram, we recognize one or more *connectors* which belong to the following set:

$$C = \{\lfloor \quad \rfloor \quad \perp \quad \top \quad \vdash \quad \dashv \quad +\}$$

Considering specific pairs of connectors, in the set C, it is possible to define

a *connection relation* (CR) between them: each pair in the relation indicates the closing of a disjunction if found in the same line of a ladder diagram.

$$CR = \{(\lfloor, \rfloor), (\bot, \rfloor), (\lfloor, \bot)\}$$

The algorithm uses two auxiliary functions, namely *LogicAnd*, which navigates the main line of the ladder diagram grouping from right to left the consecutive contacts in a single *and* gate (see the example in Figure 4), and *LogicOr*, which builds an *or* gate from the contacts that are found inside a CR pair.

Following the algorithm, the variables are extracted first from the matrix by the GetVariables function (*line* 1).

Then, the algorithm checks if the value corresponding to the symbol $\top$ is present in the equation matrix M (*line* 2); this means that at least a logic *or* is present in the ladder diagram.

If the symbol $\top$ is present the *LogicAnd* function is called to build the *and* gates from consecutive contacts on the main line of the ladder diagram (*line* 3).

Then, the algorithm executes the main loop (*lines* 4-7) in search of disjunctions from right to left, inside a window of increasing size i, by repeated calls to *LogicOr*. In each cycle, the *LogicAnd* function groups again the left consecutive conjunctions.

In the case the symbol $\top$ is not present, all the variables are in a logic *and* and the corresponding gate is built by a single call to *LogicAnd* (*line* 9).

The LLD Parser builds a combinatorial logic network that represents the right hand expression of an equation. Depending on the kind of *coil* used as a left-hand, different uses are made of the output result of the network. In particular, it may be used as:

- input to another equation
- input to the same equation
- activation of a timer

In the first case a connection with the related input to the model of the other equation is made; in the second case a delay block is inserted in the connection back to the input; in the latter case a timer is inserted (see Figure 5). This is done for all the models generated for the equations, completing

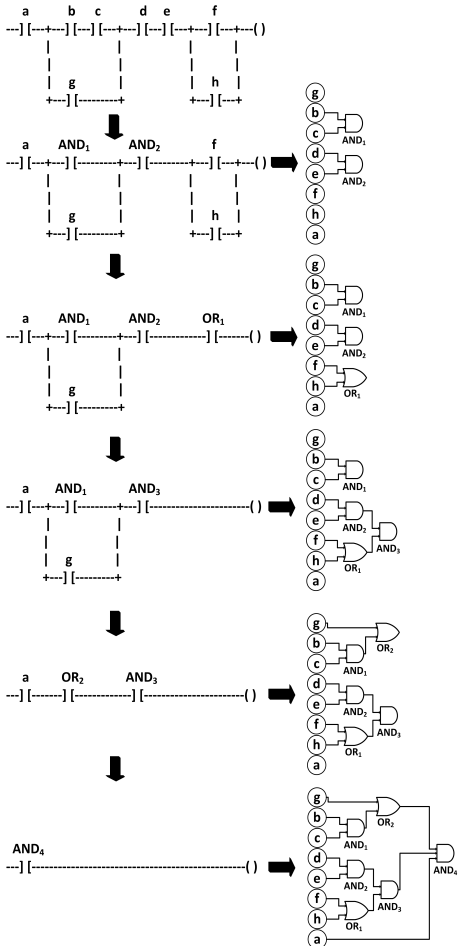

Figure 4: Model building steps

the model of a station. The model extraction procedure has been applied to equipments containing up to 12500 equations with 800 input variables. This means that the produced Simulink models are very intricate and actually not human-readable.

3.5. Use of the model - Safety requirements

The safety requirements for an interlocking system are normally expressed in terms of routes, since the aim of an interlocking is to establish safe routes through the track layout of a station. A route is intended as a set of consecutive track elements (track circuits, points, etc.) that have to be reserved

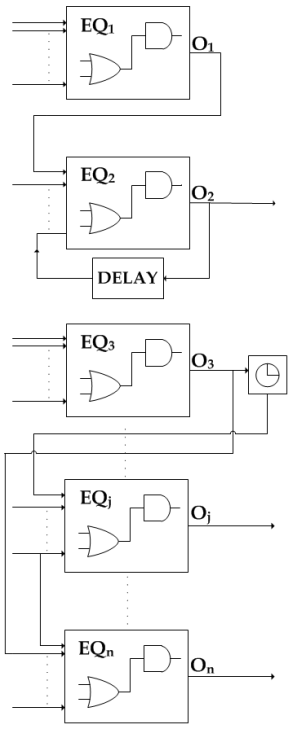

Figure 5: Example of station model

together to allow for the movement of a train through the route. In general, routes can be classified in main routes and shunting routes, protected by different signals and with different interlocking rules; we do not apply this distinction in the following. Typical safety requirements can be classified in a few classes as:

- **Route protection** *When a route is reserved for a train, appropriate means are in place to prevent another train to enter the route.*
  This requirement may be enforced by the use of different mechanisms:
  - *Entry Signals*: the aspect of an entry signal tells the driver whether the train is authorized to enter a route;
  - *Speed Signals*: the aspect of a speed signal tells the maximal speed at which the train is authorized to enter a route;
  - *Automatic Train Protection*: an on-board system receives the *movement authority* from the interlocking and automatically brakes

the train in case of missing authority

- *Overlap*: a safety zone of a certain distance after the route exit signal, used to prevent a hazardous situation from occurring if a train is unable to stop in front of the exit signal;
- *Flank Protection*: points and signals not belonging to the route are properly set in order to avoid hostile train movements into the route.

For example, in the simplest case where only the first item applies, this requirement can be expressed as:

1. *an entry signal to a route can be green only if a route protected by the signal is reserved*;
2. (to avoid unintended movement of a following train) *a green entry signal shall be set to red as soon as a train has entered the reserved route.*

These requirements should be fulfilled for every route with its protecting signal (note that one signal may in general protect more than one route)

- **No-collision**
  Once *route protection* is established, two trains cannot collide if the following holds:

  3. *Two routes that share a track element cannot both be reserved at the same time.*

  This requirement should be fulfilled for every pair of conflicting routes (that is, routes that share a track element)

- **No-derailment** *While a train is crossing a point, the point shall not change its position.*
  Due to *route protection*, this requirement can be expressed as:

  4. *While a route is reserved, any point on the route shall not change its position.*

  This requirement should be fulfilled for every pair of route and point on the route.

- **Cancellation and release** Cancellation of a reserved route occurs at a specific request from the operator. Release of the free track elements of a route occurs after the train occupying the route track elements, a connected group of elements or a complete route, has moved out of them. To avoid collisions after a cancellation or release, the following requirements shall be satisfied:

    5. *Cancellation of a route can occur only if no train has yet entered the route,* (that is, no track element of the route is occupied)
    6. *Release of track elements can occur only after the train has left the track elements.*

We can see that the generic safety requirements refer to specific layout of physical or logical entities (points, track circuits, signals, routes,...). Hence expressing actual safety requirements needs a detailed knowledge of the track layout and of the interlocking rules actually applied.
The model extraction process presented in Sect. 3.2 is actually a reverse engineering process, in which some domain knowledge is needed to relate the variables used in the equations to the controlled track layout. Due to independence constraints between the validation team and the design team, this knowledge is however not always readily available. One piece of knowledge available to the S&V team is the following convention for the variables'identifiers:

- the identifiers unambiguously indicate the kind of element to which the variable refers;
- each variable identifier refers either to a specific track layout element (track circuit, point, etc.) or to a specific route request:
    - in the former case the identifier includes the number of the referred track element;
    - in the latter case the identifier includes a number that identifies the route.

Adjacency of layout elements is inferred from the equations, rather than known in advance: the relation between the numbers contained in the identifiers and the position of the named elements in the track layout is only partially known, but it could be in principle retrieved from the equations using

the above assumptions. For example, an equation used to set the reservation of a route will have in its right hand (possibly transitively through other equations) variables related to the status of all the adjacent elements that belong to the route.

Actually, for verification purposes, it is not needed to fully retrieve the information about all track elements and routes, but in Sect. 5 we show how the knowledge of this convention turns to be useful during model checking based verification.

Indeed, the validation w.r.t. safety requirements is routinely performed by the S&V team using model-based testing over the extracted model (see Sect. 4). Furthermore, formal verification is being currently experimentally applied, in view of a full adoption in the validation process (see Sect. 5).

4. Model-based Testing

The best practice in *model-based testing* is to run a test suite on a model that is derived from the system requirements and the test suite has been obtained from the model itself (37). In this work, we rather consider an extended meaning of the term *model-based testing*: due to the particular constraints of the considered validation activity, we execute simulations of the model, using as simulation scenarios the already available test cases intended for the final system. This differs from canonical model-based testing, but we wish to maintain also this kind of activity under the umbrella of model-based testing. We refer to (29) for a more canonical model-based approach to test interlocking systems, where tests are derived from formal models derived by system requirements.

In our case, the reasoner engine, considered as the generic application according to the CENELEC terminology, is taken as already validated, and hence the focus is on validating the configuration data, that is, the control table.

For this reason, the model is not derived from the system requirements but it is directly derived from the binary files, that represent the control table as loaded on the target, by means of a *reverse engineering* process.

In the following we describe the test scenarios and the part of the verification framework which allows to simulate them. Then, we discuss the obtained results and the assessment process of the framework.

4.1. Test Scenarios

We show the results of the execution of some sample test scenarios on the models of three small stations. The models have been built using the process and algorithm discussed in the previous section. They exhibit the number of equations, inputs, outputs, gates (only *and gates* and *or gates*) and time to generate the model reported in Table 2. The last column gives, in seconds, the time needed for extracting the model.

Table 2: Stations Dimensions

Station	**Equations**	**Inputs**	**Outputs**	**Gates**	**Time**
$STATION_1$	2625	717	915	17934	2177
$STATION_2$	2090	351	437	15571	1712
$STATION_3$	3281	490	663	18001	2332

The test scenarios were identified in accordance with signalling engineers. The test scenarios are written in a tabular format in a spreadsheet where each *test input* is marked by the keyword *SET* followed by the name of the input and the corresponding value, and each *expected output* is marked by the keyword *VERIFY* followed by the name and the value.

Table 3 shows an example of a test scenario, related to a successful route request, for route 123. The route includes the points 75 and 42. The scenario aims to check that a no-collision property and a no-derailment property (see Sect. 3.5) are satisfied. The former property is verified by checking that the protection signal 35 is red, the latter by checking that the points are locked in the desired position.

After all the verifications are completed, the signal 57 authorizing transit of a train on the route is finally set to green. The interlocking rules for setting a signal requires a further verification that the signal is confirmed to be actually set at the desired value.

To simulate the test scenarios a Matlab function, named *Digital Inputs Generator* (DIG), has been developed. This function implements three phases:

1. Each test is provided to the function in a spreadsheet format. The function reads the spreadsheet and generates a set of digital inputs corresponding to test inputs (Figure 6).

Table 3: Example of Test Scenario

Keyword	**Variable Name**	**Value**	**Description**
SET	$RequestR123$	1	Route Request
$VERIFY$	$StatusR123$	1	Verify Route Status
SET	$PosPnt75$	0	Command Normal Position
SET	$PosPnt42$	0	Command Normal Position
$VERIFY$	$StatusPnt75$	0	Verify Position point
$VERIFY$	$StatusPnt42$	0	Verify Position point
SET	$SignalTL35$	0	Set Red Light
$VERIFY$	$StatusTL35$	0	Verify Red Light
SET	$SignalTL57$	1	Set Green Light
$VERIFY$	$StatusTL_{57}$	1	Verify Green Light

2. The function launches a simulation and Simulink reads the signals (*Input Digital Signals*) and produces the output values of the model (*Output Digital Signals*).
3. The values obtained by the test simulation are checked with the expected output reported in the spreadsheet and a report is generated. In the case the values of digital outputs do not fit the expected outputs in the spreadsheets, one of the following three cases has occurred:
   a. either an error is present in the control table,
   b. or an incorrect variable value has been written in the test scenario,
   c. or an incorrect station model has been generated[2].

4.2. Results

The entire process of importing data from the binaries, modelling the station and test simulation has been run on an Intel(R) Xeon(R) 3.30GHz, 128GB of RAM machine with Windows 7, 64 bits, operating system.

[2]Generation of an incorrect station model can occur only due to a bug in the LLDParser or in the legacy libraries; this possibility actually occurred during the tool testing phase, triggering appropriate corrections of the affected software components.

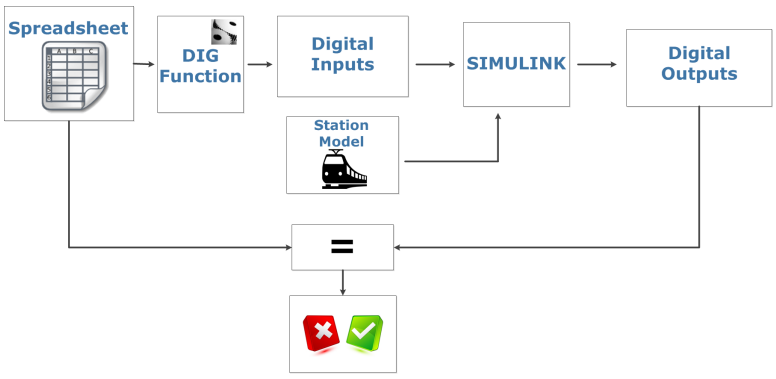

Figure 6: Simulation Framework

In Table 4, we report a comparison between the simulation time for the test scenarios run for the three stations and the respective execution time on the target inside of the physical test bed. In each case the number of executed tests (*N. of Test*) is presented, together with the mean test target execution time in seconds ($MTTET$), the mean scenario simulation time ($MSST$) with the same test scenarios, and the ratio between MTTET and MSST.

Table 4: Times

Station	**N. of Tests**	**MTTET**	**MSST**	**Ratio**
$STATION_1$	85	808.814	44.794	18.056
$STATION_2$	51	1007.946	54.615	18.455
$STATION_3$	58	1270.769	69.294	18.389

These data show that simulation appears to be around 18 times faster than testing the target, hence allowing for testing cost reduction.

4.3. The certification process

The certification according to safety guidelines can hardly allow not to run the tests on the final system. Early simulation of test scenarios can however avoid hours of target testing rework, due to possible early discovery of bugs in the description of interlocking rules or of inaccuracies in the test suite.

The above results have therefore convinced GETS to set up a test framework that includes the following features:

- extraction of a Simulink model from the implemented control tables;
- adaptation of the incoming test suite to be used as a scenario simulation for the Simulink model;
- run of the simulation and logging of results, to be eventually compared for equivalence with the corresponding test run on the target.

The framework has been subject to an external safety assessment process by an assessor (ASR).

To approve the framework the ASR requested: (1) the description of each Matlab function, (2) the dependency graph of the Matlab functions and (3) the user manual of the framework. Moreover, the ASR requested a *demo* of the framework; the demo imported and modelled a station and executed some test scenarios.

After this process the framework has been approved for stations with up to 5000 equations and it is currently included in the standard validation process.

4.4. Framework Enhancements

Currently, the framework has been enhanced to consider the interlocking logics of stations of about 7000/8000 equations, of a railway warehouse with about 12500 equations and of a so-called multi-station interlocking, controlling a line inclusive of some adjacent stations (about 9200 equations).

In the latter case, a compositional method has been implemented for the model generation. Indeed, the overall line model is obtained by linking station models by means of their interfaces at their borders, where the interfaces consist of variables that give the status of the traffic on the line.

The proven capacity of the framework on larger size interlocking systems has triggered a supplement of safety assessment, requiring new documentation to be sent to the ASR, in order to justify its usage of the framework up to these larger size figures.

5. Model Checking

In recent years model checking techniques have raised the interest of many railway signalling industries being the most lightweight method, from the process point of view, and being rather promising in terms of efficiency: safety properties of an interlocking system are quite directly expressed in

temporal logic, and their specifications by means of control tables can be directly formalized (7; 19; 23; 25; 26; 30).

Control tables may indeed play two main roles (not always both present) in the development of these systems:

- as specifications of the interlocking rules (20; 22), often issued by a railway infrastructure company;
- as an implementation means, when they come encoded in some executable language, that may be either proprietary or standard, as is the case for ladder diagrams.

In the first case, verification of control tables may address self consistency of the specification, or correctness of the implementation w.r.t. the specification, while in the second case it may be focused on the check of safety properties (expressed for example in a temporal logic) on the implementation. Typical issues of any of these verification tasks is the combinatorial state space explosion problem, due to the high number of boolean variables involved, and the choice of how to express control tables in a language suitable for the verification tool adopted.

The first applications of model checking have therefore addressed portions of an interlocking system (3; 18); but even recent works (17; 38) show that routine verification of interlocking designs for large stations is still a challenge for model checkers, although specific optimizations can help (38).

5.1. Satisfability Problem for Interlocking Systems

A recent effort has collected up-to-date reports about interlocking verification (7; 19; 25; 26). Notwithstanding their different verification aims, one point in common to these works is the use of SAT-based model checking, which appears to be more promising due to the native boolean coding of the control tables.

Indeed, according to the representation given in Sect. 3.1 of the set of boolean equations we can call $\tilde{x}_i, \tilde{y}_i$ the vectors of values taken by such variables in successive executions. From the equations we can define $F(\tilde{x}_i, \tilde{x}_{i+1}, \tilde{y}_i)$ as a boolean function that is true iff $\tilde{x}_{i+1} = f(\tilde{x}_i, \tilde{y}_i)$, representing one execution of the equations. Let $Init(\tilde{x})$ be a predicate which is true for the initial vector value of state and output variables. If $P(\tilde{x})$ is a predicate telling

that a desired (safety) property is verified by the vector $\tilde{x}$, then the following expression:

$$\Phi(k) = Init(\tilde{x}) \wedge \bigwedge_{i=0}^{k-1} F(\tilde{x}_i, \tilde{x}_{i+1}, \tilde{y}_i) \wedge \bigvee_{i=0}^{k} \sim P(\tilde{x}_i)$$

is a boolean formula that tells that P is not true for the state/output vector for some of the first k execution cycles.

According to the Bounded Model Checking (BMC) principles (4), using a SAT-solver to find a satisfying assignment to the boolean variables ends up either in unsatisfiability, which means that the property is satisfied by the first k execution cycles, or in an assignment that can be used as a counterexample for P, in particular showing a k-long sequence of input vectors that cause the safety problem with P.
SAT-based Bounded Model Checking can be complemented by *k-induction* (32; 12) to show satisfiability beyond the set bounds, and may be made more efficient by employing SMT (Satisfiability Modulo Theory) solvers (1).

These techniques are applied indeed in commercial solutions for the production of interlocking software, such as Prover Technology's Ilock (24), that includes formal proof of safety conditions as well, by means of a SAT solving engine. Industrial acceptance of such "black-box" solutions is however sometimes hindered by the fear of vendor lock-in phenomena and by the loss of control over the production process.

In our case, however, the interlocking design process is not an issue: the S&V team receives already encoded control tables in a target machine, and the extracted Simulink model is therefore the input to the model checking activity.

The clear preference of GETS towards commercial tools, together with the natural integration with Simulink and its SAT-solving engine has suggested the adoption of Matlab *Design Verifier* (33).

Notwithstanding the interesting performances of Design Verifier on large models, state explosion on thousands of equations is overwhelming. Hence, strategies to contain the state space are eagerly needed.

5.2. Environment assumptions and Slicing

We can observe that the state space of a model of an interlocking system depends on the modelling of its environment as well. For example, the study of (17) made no assumption whatsoever on the environment: the study was

aimed at finding the limits of verification of a completely unconstrained set of boolean equations, with little attention to realism of the equations set w.r.t. actual interlocking rules. The particularly negative outcomes of that work in terms of size of tractable interlockings were mostly due to the absence of constraints on the external environment, that is to the assumption that the system is open to any behaviour of the environment.

Most works on formal verification of interlocking systems do instead make assumptions on the behaviour of the environment, in order to constrain the state space, but also because some verification frameworks can only deal with a closed system. Such assumptions may take the form of an explicit model of the environment: for example, trains moving on the controlled track layout are also modelled, and the trains often obey to some reasonableness constraint, such as trains moving in only one direction, appearing in the layout only at its borders, respecting signals, and so on. Such constraints enforce only particular sequences of events (e.g. track occupancy events) to be possible inputs for the interlocking systems, and consequently limit the state space explosion typical of when considering a fully open environment. It is then a matter of the safety assessment process to demonstrate that the properties proved under given assumptions are maintained in any real situation due to the reasonableness of the environment. For example, it is possible in many cases to show that modelling two trains is enough to cover cases with more trains present in the track layout (14; 31).

Such assumptions in general refer to local properties, that is, for example, *no-derailment* on a point is scarcely related to the location of a distant point on a parallel track. To be more precise, locality is implied by the definition of *routes*, that is, the set of contiguous track elements that need to be granted for a given train movement: the routes insisting on a given track element define the elements that may be directly related to the status of the given element. Locality given by the topological layout of the controlled systems has been used in (38; 39; 40) to define domain-oriented optimizations of the variable ordering in a BDD-based verification.

Locality can be used also for slicing, as suggested in (17) and (25). The idea is to consider only the portion of the model that has influence on the property to be verified, by a topological selection of interested track elements: this allows for a much more efficient verification, at the price of repeating the verification activity for each extracted slice and of showing that verifying slices does implies the satisfaction of desired properties for the whole system. Extracting a slice of the model implies to make assumptions on the environ-

ment of the slice: either an open, unconstrained environment, or constrained by reasonableness assumptions. Verification of a slice is targeted therefore to the satisfaction of local properties of the slices, under the assumptions (possibly none) given for the environment of the slice. It is therefore needed to show that the satisfaction of local properties under the given environment assumptions imply the satisfaction of global properties of interest.

Notice that in our case the term *slicing* gets a slightly different meaning with respect to the consolidated definition of program slicing (36). As in program slicing, the slice is built, starting from a set of variables, by considering only those statements (in our case, equations) whose execution may affect the value of variables. While in program slicing this just removes irrelevant steps, producing a *precise approximation* of the behaviour, in this case it also removes references to parts of the external environment by removing tout-court some constraints on it, leading to an *over-approximation* of the behaviour. So, we use the term *slicing* to refer to the syntactic technique to produce a property-preserving abstraction of the system.

We have already described in Sect. 3.5 the naming conventions that hold for variables in our equations: each track element n, where n is a unique identifying number, is associated to several variables whose name contains the number n; for example *ND35* and *NC11-77*. This guides the extraction of a slice. Consider for example a no-derailment property that can be expressed as: by no way point 35 can move while track circuit 18 is occupied. A slice can be built by considering only the equations that include variables whose name contains 35 and 18. All the other variables in the right hand of the equations are considered as free variables, and hence constitute the environment of the slice, either open or constrained by some reasonableness assumptions. If it comes out that the verification of the property on the slice/environment pair does not give a clearly positive answer, the pair should be refined in a successive step, giving rise to an iterative verification process.

This reasoning can be extended to any safety property. Consider a safety property ϕ that tells that a dangerous situation referring to the status of some elements $\{t_1, t_2, \ldots t_n\}$ is never reached. Let us build the smallest slice M' selecting only the equations with those variables $\{x_1, x_2, \ldots x_m\}$ that in their identifiers refer to $\{t_1, t_2, \ldots t_n\}$. Let us call $\{y_1, y_2, \ldots y_p\}$ all the other variables of the selected equations. In this slice all the equations assigning a value not depending from any x_i to a variable y_j is omitted. The values taken by the y_j variables are not defined, hence they can be considered as part of the fully open environment. The states reachable by executing the slice

M' with any input are a superset of the states reachable when we add any constraint to the values taken by the y_j variables (e.g., by adding equations not present in M', or constraining the environment). Hence, if $M' \models \phi$, then the property is satisfied by the whole equation system M. Otherwise, a counterexample for $M' \models \phi$ may tell either that the property is not satisfied at all by M, or that M' has too few equations, or that the environment is not sufficiently constrained.

5.3. CEGAR-like verification process

In (6) we have proposed an iterative verification process inspired by the CEGAR (CounterExample Guided Abstraction Refinement) paradigm (11), in which the analysis of counterexamples drives the refinement of the model for a further verification cycle. the process is represented in Fig. 7.

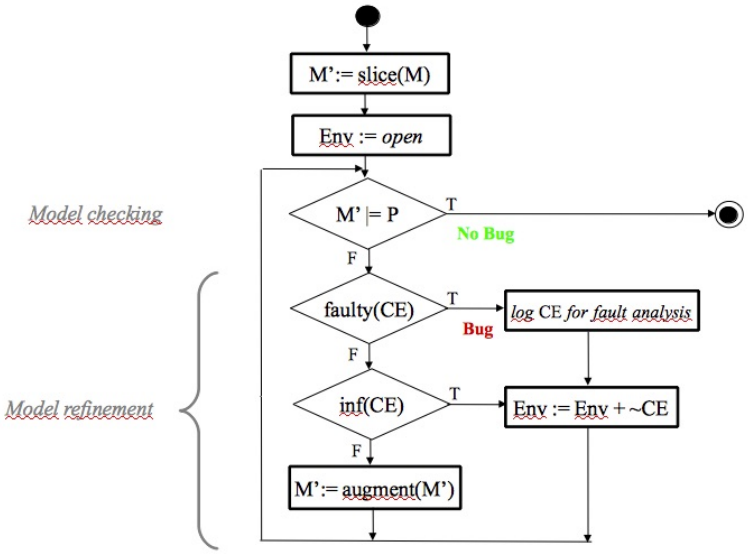

Figure 7: CEGAR-like verification loop

The initial slice M' is extracted from the input model M on the basis of the required property P, focusing on the set of track elements or routes, distinguished by a unique numeric identifier, that are referred by P. Hence the slice is constituted by the equations that refer to variables having those numbers in their identifier. All the other variables in the right hand side of

the equations are considered as free variables, and hence constitute the fully open environment of the slice.

If the property P is not verified, a counterexample CE is generated: the counterexample is examined in order to refine the slice or its environment to remove the occurrence of spurious counterexamples (*model refinement step*). This step can exploit different techniques, depending on the results of the analysis conducted on CE. The counterexample analysis is strictly dependent on the functional and safety aspects of system at hand, and hence requires some knowledge of the produced system: it is therefore a mostly manual analysis requiring help from signal engineers.

First of all, the analyst can decide that CE is actually related to some fault. The verification process can continue after storing the faulty scenario. The data that have produced the counterexample are then excluded from the next verification step, typically by constraining the environment by means of the negation of CE.

In the case CE is not considered to exhibit a fault in the model, the analyst has to distinguish counterexamples that can be clearly decided to be infeasible due to, e.g., physical constraints, from those that may be possibly unfeasible due to the actual behaviour of the environment of the slice, which is constituted by the equations excluded from the slice. In the first case, it is likely that the counterexample is generated by an unfeasible combination of values of input variables, while in the second case it is due to the assumption of unconstrained behaviour of adjacent elements, that is, by an unfeasible combination of values of latch variables that are assigned by equations which have been excluded from the slice.

In the first case the analysis continues on the same slice, by excluding the input data that has produced CE, while in the second case the slice is considered too small to continue the verification process, and is augmented by bringing in all the equations that contain free variables of the M' slice[3]. Recall that at the first step the free variables of M' are those that do not contain the identifiers of the track elements on which the slide was first built: hence this step enlarges the scope of the considered slice to some other track elements, that with high probability are physically adjacent to the original ones due to the locality principles.

[3]This step can be automated, although in the experience reported in Sect. 5.4 it has been performed manually.

Usually the property is invariant w.r.t. the enlarging of the scope of the slice, since it refers to variables that are still part of equations in the augmented slice. In some cases, it might be needed to refine the property for consistency with the slicing mechanism. In this case, the refined property P' to be used in the next verification step should be such that $augment(M') \models P'$ implies that M satisfies P when embedded in its proper environment.

The model refinement cycle terminates when the property is verified (possibly trivially because the model has become over-constrained): at that point the possible faulty counterexamples stored during the process can be subject to further analysis in order to plan corrective action or to support fault analysis conducted at the system level.

We finally note that the presented process is independent from the model checking tool used: in the following we discuss some example verification results achieved with Design Verifier.

5.4. Verification with Design Verifier

5.4.1. No-derailment property

We show in the following some example verifications performed on computer-based interlocking subsystems that control small railway stations. The entire process of importing data production binary files, modelling the station and proving the properties has been run on a DELL XPS L501X 2.67GHz, 4GB of RAM machine with Windows 7, 64 bits, operating system.

In the first case, the subsystem has 1038 equations, 321 inputs and 470 outputs; each equation can have from one input to a maximum of 25 inputs. The size of the model is therefore rather large, and slicing is therefore considered according to the iterative process defined in section 5.3. The verification considers a *no-derailment* property *ND35* for a chosen point, numbered 35 (referring to the fragment of track layout represented in Fig. 9). The property is defined as:

Property *ND35*

Under the preconditions (which are considered to characterize the state in which the route is granted for the passage of a train):

1. A *route request* for the route 11, that includes point 35, has been received.
2. In accordance with the route request the points belonging to the route are in the correct position.

3. The route is reserved.

the point 35, in the case a *movement request*, *MR35*, is received, must not move either in normal position, *NPM35*, or in reverse position, *RPM35*.

The verification with Design Verifier requires the property *ND35* to be expressed as a Simulink "observer", as shown in Fig 8.

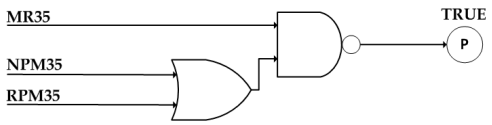

Figure 8: Representation of the property *ND35* in Simulink

According to the process described in Section 5.3, we generate the smallest slice of the computer-based interlocking subsystem to prove the property *ND35*; this slice includes all the equations referring to variables whose identifier contains the number 35: the slice has 16 equations and 70 inputs. The slice also assumes that track circuits 18 and 23 (which are shared by route 11 and 82) are initially not occupied.

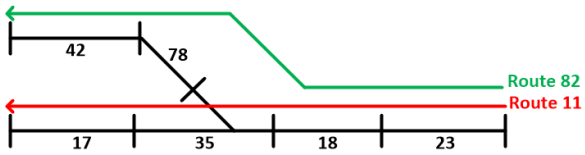

Figure 9: Fragment of station plan for the no-derailment property

The verification on $Slice_1$ has generated 15 counterexamples (with 15 CEGAR iterations) before satisfying the property. Each counterexample has been excluded with the introduction of an environment assumption on input variables. When the property was satisfied, the analysis of the environment assumptions, introduced by the previous generated counterexamples, has shown these other preconditions:

a. No other route request must have been received.
b. The route reservation announces to the adjacent station on the route the next arrival of a train, and the station acknowledges this announcement: this acknowledgement must be already received.

This consideration has suggested that enlarging the slices as recommended in the iterative process would have soon given better results. Indeed, we have performed such enlargement ($Slice_2$) that has allowed us to include such pre-conditions in the slice itself. The new slice contains all the equations that control the track circuits 17, 18, 42 and 78; hence, the boundary of the slice towards the environment has moved from track circuit 18 to track circuit 23, and so have done the related assumptions. The verification of the same property for this slice has immediately produced no counterexamples.

In order to test the sensitiveness of the approach to timing issues, we have included in $Slice_3$ a timer equation present in the original model, but which was not included in $Slice_2$. The duration of the timer in the model is set to 5 seconds, while the DV parameter called *simulation step* is 150 milliseconds: modelling the timer amounts to wait for a sufficient number of simulation steps. With this value, the verification was not concluded in reasonable time (presumably, the property was satisfied, but DV was not able to complete the search for non-existing counterexamples) because it needed too many simulation steps, which increase the state space size beyond reasonable limits. To achieve the verification of the property we adopted a different time scale, by changing the *simulation step* to half a second, separately checking that the chosen different time scale was still compatible with the functional behaviour of the slice. The property is verified with the compressed time scale, and the computation time has been dramatically decreased, well beyond expectations.

In Table 5 we report for each slice the number of equations and the number of free inputs, the number of counterexamples produced before proving the property, the simulation step used and the time taken by Design Verifier to complete a single step of the verification loop.

Table 5: Verification Results

Slice	**Equations**	**Inputs**	**Counterex.**	**Simul. Step (ms)**	**Time (sec)**
$Slice_1$	16	42	15	150	60.0
$Slice_2$	64	94	0	150	104.0
$Slice_3$	65	94	0	500	46.0

5.4.2. No-collision property

We show the verification of a *no-collision* property as well, *NC11-77*, which refers this time to the route requests for the routes 11 and 77 on the same track layout fragment (see Fig. 11):

Property *NC11-77*

Under the preconditions:

1. A *route request* for the route 11 has been received.
2. The route is reserved.

the route request for the *conflict* route 77 must not be accepted.

The verification with Design Verifier requires the property *NC11-77* to be expressed as a Simulink observer, as shown in Fig 10.

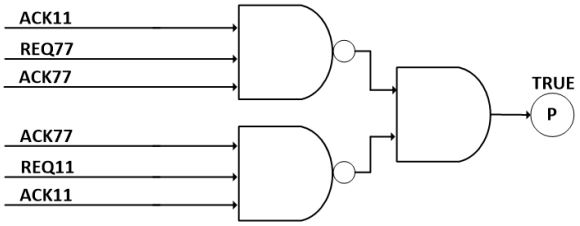

Figure 10: Representation of the property *NC11-77* in Simulink

A different slice, with 29 equations, has been modelled to prove the property *NC11-77*, which has been verified without producing any counterexample in about 30 seconds.

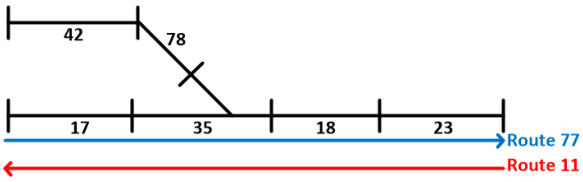

Figure 11: Fragment of station plan for the no-collision property

5.4.3. Automatic Cancellation

In the end, we show the verification of a more complex property regarding the *automatic cancellation* feature, (also known as *sequential release*(28))

that releases an element in a reserved route as soon as the train has passed it. Safety for automatic cancellation is guaranteed when in no way the cancellation is activated before the train has passed the relevant elements. The examined property, that refers high level variables (that is, variables which encode high level commands and situations, not the status of track elements) is defined for route 72 as:

Property *AutomCanc*

Under the preconditions (which are considered to characterize the state in which a train is going to occupy a route granted for it):

- a *route request* for the route 72 has been received and the route is reserved (precondition represented by the variable RR72);
- a train is approaching the entering track of the route (precondition defined as an environment assumption).

then in no case the automatic cancellation of the route should be activated (AC72).

The Simulink observer for the property is elementary, as shown in Fig 12.

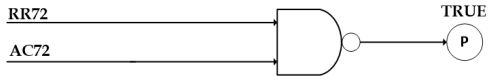

Figure 12: Observer for the property AutomCanc

In the verification of this property we have experimented the use of the Mathworks *Model Slicer* toolbox. Model Slicer performs a slicing on the Simulink model, starting from the output variables that are referred to in the property, and navigating backwards the model data flows to retrieve all the Simulink blocks influencing the property. We call in the following a slice generated by Model Slicer as *property-driven*, while slicing obtained through looking at variable identifiers as described in Sect. 5.2 is called *layout-driven*. It turns out that the layout-driven slice is a sub-model of the property-driven slice, because it does not transitively considers other variables influencing the property (that are considered as open environment), but only the ones with the related identifiers: only through a model refinement step the slice is gradually enlarged to eventually coincide with the property-driven one. The

initial model had 930 equation and the *Model Slicer* generated a property-driven slice ($Slice_1$) with 290 equations. The layout-driven slice ($Slice_2$) has only 3 equations and is represented in Figure 13).

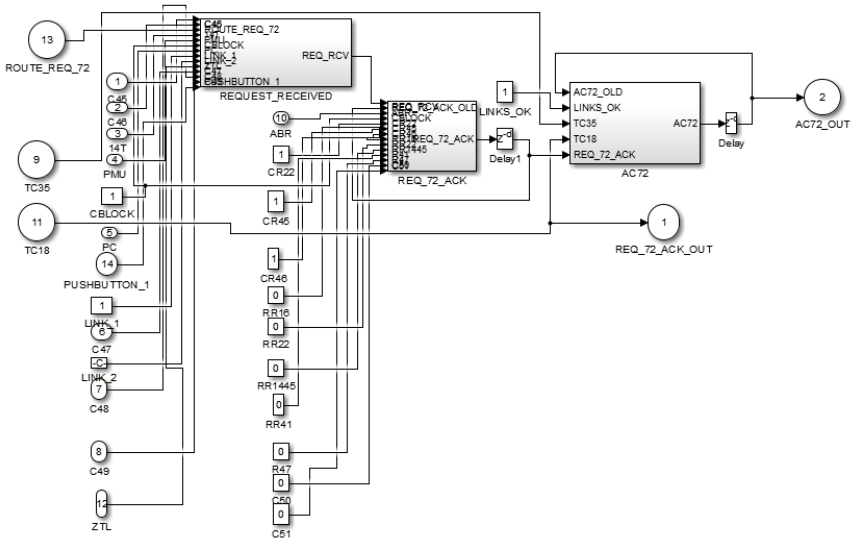

Figure 13: Minimal slice to prove AutomCanc

A first counterexample has been generated by Design Verifier for both slices. The signalling engineers classified these counterexamples as unfeasible and hence *spurious*, since the counterexamples showed a train which moved in a route in conflict with route 72, and no-collision was already proved for the two routes.
After the environment has been modified to exclude that counterexample, a further counterexample has been generated, as shown in Table 6, exhibiting an (unexpected) occupation in sequence of the point 35 and track circuit 18 (see Fig. 14), that does not prevent automatic cancellation to be issued. Signalling engineers have considered that such an unexpected occupation is possible in some failure cases, and issuing automatic cancellation in such cases is an actual safety threat which had not been detected during previous testing activity, hence triggering a corrective action on the control tables.

In Table 7 we show the metrics (number of equations and inputs) of the slices for this property and the time to find out the counterexample, that indicates that the (domain-specific) layout-driven slicing mechanism is more efficient in giving the same results.

Table 6: Counterexample

Step	**1**	**2**
$TC35$	0	0
$TC18$	1	0

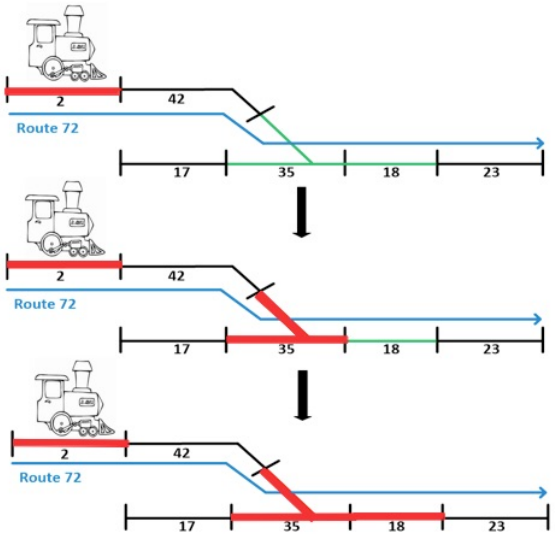

Figure 14: Unexpected occupation counterexample

5.5. Performance of Design Verifier

To evaluate the performance gain obtained by slicing, we report in Table 8 the results related to the verification of the *ND35* property on slices of growing size. Fig. 15 represents the actual station plan (with some abstraction from the real one for confidentiality reasons): the station is actually controlled by three computer-based interlocking subsystems (named CIS_1, CIS_2, CIS_3), on which the set of equations constituting the station logic has been split. Inside the area controlled by CIS_2, we can recognize the fragment of the layout that we have considered above: starting from $Slice_1$, we can iteratively apply the process used above to augment it to $Slice_2$, until the border of CIS_2 is reached: it can be seen that the slice we obtain, $Slice_{max}$, will contain only some of the equations of CIS_2. We consider hence proving the *ND35* property on $Slice_{max}$, on the whole set of equations of CIS_2, and on the whole station logic, expecting that it is

Table 7: Performance of DV on property *AutCanc*

Slice	**Equations**	**Inputs**	**Time (sec.)**
$Slice_1$	290	2	281
$Slice_2$	3	2	3

verified on all these models, for the locality properties and for the fact that it has been verified by $Slice_2$.

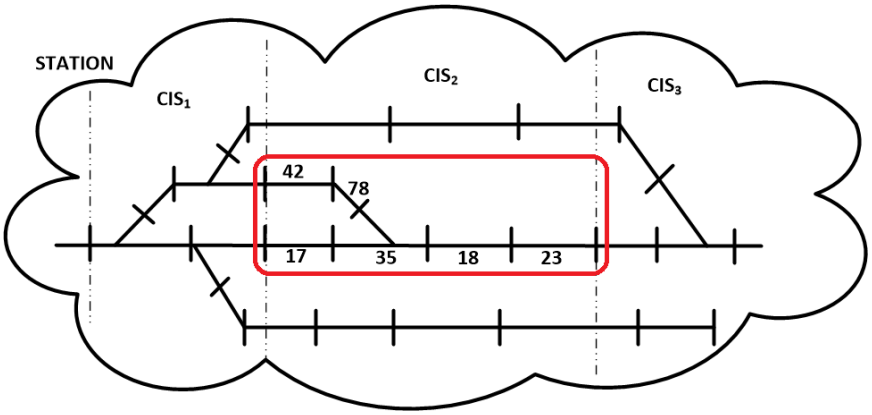

Figure 15: The complete example station plan

The *ND35* property is proved correct for $Slice_{max}$ in about 13 minutes and for CIS_2 in about 2 hours and 45 minutes (see Table 8). The proof of this property on the whole station model raised unresolved model size problems right at the start of the proof.

Slicing is hence vital for the actual application of automated formal verification to interlocking systems controlling medium to large sized stations.

5.5.1. Verification Depth

The above results have been obtained with the default value of 20 *simulation steps* for the Design Verifier parameter, called *violation steps*. This parameter indicates the depth of the state space (13) computed by the Bounded Model Checker; this means the Bounder Model Checker computes with maximum depth of 21 states.

We occasionally used the strategy described in the Design Verifier User Guide aiming at proving a property for greater depth values:

Table 8: Verification Results on Slice_{max}, single CIS and the entire station

Slice	**Equations**	**Free Inputs**	**Counterex.**	**Sim. Step (ms)**	**Time (sec)**
$Slice_{max}$	237	128	0	500	775.0
CIS_2	1608	266	0	500	9844.0
$STATION$	2625	602	-	500	-

- Set the *Strategy* parameter to *Prove* and set the *Maximum Analysis Time* parameter to 5-10 minutes.
- Set the *Strategy* parameter to *FindViolation* and set *Maximum Violation Steps* parameter to 4,5 or 6 steps.
- If you do not find any short counterexample:
  - increase the bound for *Maximum Violation Steps* (we increased it to 30, a value that it has been shown sufficient for our purposes).
  - switch back to *Prove* strategy, and use a longer time limit, such as several hours, to find long counterexamples.

5.5.2. Alternative model extraction possibilities

The proprietary nature of Design Verifier does not allow to tune its SAT-solving procedures for improving the verification performance. The only opportunity left to optimize verification was trying to use different modelling strategies, and comparing their verification performance. We have considered the three alternatives given by Simulink to express boolean functions:

- Simulink logical gates blocks, as described in previous section;
- Simulink blocks, called *combinatorial logic*, which explicitely express, for each equation, its complete truth table.
- Stateflow truth tables (34), that allow to define a *minimal truth table* by means of *don't care* conditions.

However, we have soon realized that the modelling by means of combinatorial logic blocks was not feasible due to the required full definition of the truth table; indeed, for a typical station, (with several equations with more than 20 input variables), a memory overflow even occurred already during the model extraction phase with this option. We have therefore compared the other two alternatives. In the conducted experiments, a *no-derailment* property and a *no-collision* property were considered. For both, property proving is from 1.5 to 3 times faster on the models with logical gates than on the models with Stateflow truth tables.

Apparently, starting from logical gates the internal translation of Design Verifier yields a more efficient representation in terms of satisfying assignment problem. So the initial choice has been confirmed as the best one, given the choice of the tool.

6. Conclusions

The high costs of the validation process of an interlocking system are due to the fact that the system under test is very complex and it is necessary to build a proper test-bench where the real interlocking system is reproduced.

We have reported over the industrial application of model-based testing techniques that employ a model extraction procedure. The extraction procedure generates a Simulink model that mimics the behaviour of the production target system, ruled by the implemented control tables: simulating established test suites on such a model can hardly avoid the final, lengthy validation tests on the target, due to the demanding requirements of EN50128 standard, but it has a twofold advantage: it allows for early discovery of errors in the control tables or of inaccuracies in the test suites and it introduces a form of redundancy with diversity in the validation process.

The first results have convinced GETS to adopt this procedure to test the next interlocking systems. Indeed, simulation appears to be around 18 times faster than testing the target, so that running the foreseen test scenarios in advance on the extracted models can spare future rework on the anyway necessary tests conducted on the target.

We have then shown how the same extracted model can be subject to formal verification, by means of Design Verifier, a commercial model checker based on a SAT-solver; an iterative verification process implementing slicing and CEGAR-like techniques has been defined to address the typical complexity of this application domain.

Further experience with the application of model checking techniques is needed before they can be accepted in the standard validation process in place of testing. The preliminary analysis shown in this paper has left several open issues to be addressed in future work.

- Confidence on the model-checking results: to be actually used for certification purposes in place of testing, the used tool should in principle be demonstrated not to miss any error in the model;

- Proprietary nature of the verification tool, that has not allowed to customize the verification strategy beyond those it natively provides; in particular the tuning of the verification depth has to be better analysed.
  Coupled with the previous observation, this one points to the parallel usage of a different model checker, relaxing the initial constraints set by the industrial partner on the choice of the tool. Indeed, the proposed iterative verification process is actually independent from the particular model checker used, and we have currently started to exploit such independence to create a framework that extracts the model in an intermediate format, suitable for a further translation to different model formats. NuSMV (10) is currently experimented as a target model-checker for such translation.

- Sensitiveness of the verification process to timing issues: it has yet to be understood whether the time scale compression techniques previously shown can be generally applied to other models including timers.

- Relations between the used variables identifier convention and the actual topology of the track layout, that need to be systematically identified, so that a more efficient automated slicing mechanism can be defined, in order to produce a more precise approximation to reduce the number of the CEGAR cycle iterations.

The developed framework and approach has been designed for an ad hoc industrial environment, hence it cannot be ported as it is to other interlocking producers that follow different development processes. However, we believe that the lessons learned by this experience can be useful to other manufacturers, as well as to other production contexts. In particular, we have observed that the iterative verification approach, with its incremental

nature, has helped the verifier to acquire a step-by-step increasing confidence on the system's behaviour, with targeted interactions with signal engineers. This is expected to be an added value when the approach is adopted within an independent validation division, that (often on purpose, due to the independence constraints) has not the detailed, low-level, knowledge on the product available.

Acknowledgments

The second author has been partially funded by Villum Fonden.

References

[1] A. Armando, J. Mantovani, L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. STTT 11(1): 69-83 (2009)

[2] S. Bacherini, A. Fantechi, M. Tempestini, N. Zingoni. A Story About Formal Methods Adoption by a Railway Signaling Manufacturer. In: FM '06, LNCS 4085, pp.179–189 (2006).

[3] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G., Mongardi, D., Romano. A Formal Verification Environment for Railway Signaling System Design. Formal Methods in System Design, 12(2): 139–161 (1998)

[4] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In: TACAS '99, LNCS 1579, pp.193–207 (1999)

[5] A. Bonacchi, A.Fantechi. Validation of Interlocking Systems by Testing their Models, In: QUATIC 2014, pp.226-229. IEEE, 2014.

[6] A. Bonacchi, A.Fantechi. On the Validation of an Interlocking System by Model-Checking, In: FMICS 2014, LNCS 8718, pp.226-229.

[7] A. Bonacchi, A.Fantechi, S. Bacherini, M. Tempestini, L. Cipriani. Validation of Railway Interlocking Systems by Formal Verification, a Case Study, In: SEFM Workshops 2013, LNCS 8368, pp.237-252.

[8] B. Broekman and E. Notenboom. Testing Embedded Software. Addison-Wesley, 2002.

[9] European Committee for Electrotechnical Standardization, CENELEC, EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, 2011.

[10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: CAV'02, LNCS 2404, pp. 359–364 (2002).

[11] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement. In: CAV '00, LNCS 1855, pp. 154–169 (2000)

[12] L.M. de Moura, H. Rue, M. Sorea, Bounded Model Checking and Induction: From Refutation to Verification. In: CAV '03, LNCS 2725, pp. 1426 (2003).

[13] Design Verifier, http://it.mathworks.com/products/sldesignverifier

[14] A. Fantechi. Distributing the Challenge of Model Checking Interlocking Control Tables. In: ISoLA (2) 2012, LNCS 7610, pp.276-289.

[15] A. Ferrari, A. Fantechi, S. Gnesi, G. Magnani: Model-Based Development and Formal Methods in the Railway Industry. IEEE Software 30(3): 28-34 (2013).

[16] A. Ferrari, G. Magnani, D. Grasso, A. Fantechi, M. Tempestini. Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer, IJERTCS 2(2): 42-61 (2011)

[17] A. Ferrari, G. Magnani, D. Grasso, A. Fantechi, Model checking interlocking control tables, In: 8th FORMS/FORMAT symposium, pp. 107-115, Springer (2010).

[18] J.F. Groote, S. van Vlijmen, J. Koorn.The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In: Logic Group Preprint Series 121. Utrecht University (1995)

[19] A. E. Haxthausen, J. Peleska, R. Pinger. Applied Bounded Model Checking for Interlocking System Designs, In: SEFM Workshops 2013, LNCS 8368 pp.205-220.

[20] A.E. Haxthausen, M.L. Bliguet, A.A. Kjær. Modelling and Verification of Relay Interlocking Systems. In: Monterey Workshop 2008, LNCS 6028, pp. 176-192 (2008).

[21] A.E. Haxthausen. Developing a domain model for relay circuits. Int. J. Software and Informatics 3(2-3), 241–272 (2009).

[22] A.E. Haxthausen. Automated generation of formal safety conditions from railway interlocking tables. STTT 16(6): 713-726 (2014)

[23] T. ten Hoeve. Model Based Testing of a PLC Based Interlocking System, MSC Thesis, University of Twente (2012).

[24] Ilock, http://www.prover.com/products/prover_ilock/verifier_module/

[25] P. James, A. Lawrence, F. Moller, M. Roggenbach, M. Seisenberger, A. Setzer, K. Kanso, S. Chadwick. Verification of Solid State Interlocking Programs, In: SEFM Workshops 2013, LNCS 8368, pp.253-268.

[26] P. James, F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, H. Treharne, M. Trumble, D. Williams. Verification of Scheme Plans using CSP||B, In: SEFM Workshops 2013, LNCS 8368, pp.189-204.

[27] K. Kanso, F. Moller, A. Setzer. Automated verification of signalling principles in railway interlocking systems. Electron. Notes Theor. Comput. Sci. 250(2): 19–31 (2009).

[28] L. H. Vu, A.E. Haxthausen, J. Peleska. Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release. In: FTSCS, Communications in Computer and Information Science 476, pp. 223-238 (2015)

[29] L. H. Vu, A.E. Haxthausen, J. Peleska. A Domain-Specific Language for Railway Interlocking Systems. In: 10th FORMS/FORMAT symposium, pp. 200–209, Technische Universität Braunschweig (2014).

[30] H. Löding, J. Peleska. Timed Moore Automata: Test Data Generation and Model Checking. ICST '10 pp. 449–458. IEEE Computer Society (2010).

[31] F. Moller, H. N. Nguyen, M. Roggenbach, S. Schneider, H. Treharne, Defining and model checking abstractions of complex railway models using CSP||B. HVC'12, LNCS 7857, pp. 193-208 (2012).

[32] M. Sheeran, S. Singh, G. Stålmarck, Checking Safety Properties Using Induction and a SAT-Solver. In: FMCAD 2000, LNCS 1954, pp. 108125.

[33] Simulink, http://www.mathworks.com/products/simulink/

[34] Stateflow, http://www.mathworks.com/products/stateflow/

[35] S. Vanit-Anunchai. Modelling Railway Interlocking Tables Using Coloured Petri Nets. In: COORDINATION LNCS 6116, pp. 137–151 (2010).

[36] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, 10 (4): 352-357 (1984).

[37] S. Weißleder. Test models and coverage criteria for automatic model-based test generation with UML state machines. PhD Thesis Humboldt Universität Berlin (2010).

[38] K. Winter, N.J. Robinson. Modelling Large Railway Interlockings and Model Checking Small Ones. In: Twenty-Sixth Australasian Computer Science Conference (ACSC2003) pp. 309–316 (2003)

[39] K. Winter, W. Johnston, P. Robinson, P. Strooper, L. van den Berg. Tool support for checking railway interlocking designs. In: 10th Australian workshop on Safety critical systems and software. pp. 101–107 (2006)

[40] K. Winter, Symbolic Model Checking for Interlocking Systems, in: F. Flammini, ed., Railway Safety, Reliability, and Security: Technologies and Systems Engineering, IGI Global, May, 2012